Algorithms and Data Structures 2
CS 1501

Spring 2023

Sherif Khattab

ksm73@pitt.edu

(Slides are adapted from Dr. Ramirez's and Dr. Farnan's CS1501 slides.)

Announcements

- Upcoming Deadlines
  - Homework 2: this Friday @ 11:59 pm
  - Lab 2: Tuesday 2/7 @ 11:59 pm
  - Assignment 1: Friday 2/17 @ 11:59 pm
- Lecture recordings are available on Canvas under Panopto Video
- Please use the “Request Regrade” feature on GradeScope if you have any issues with your homework grades
- TAs student support hours available on the syllabus page

Previous lecture

- Binary Search Tree
  - How to search and add
  - three cases for delete
- Runtime of BST operations
  - add, search

Today

- Binary Search Tree
  - How to delete
- Red-Black BST (Balanced BST)
  - definition and basic operations

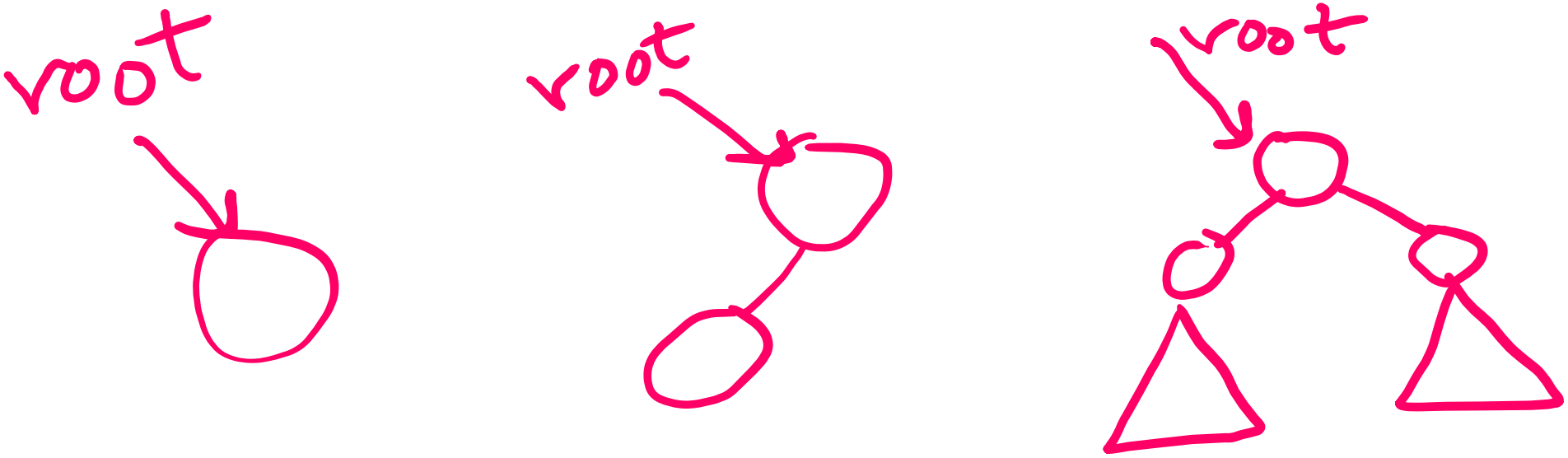
Let's see the code for deleting from a BST

- Available online at:
  - https://cs1501-2231.github.io/slides-handouts/CodeHandouts/TreeADT/Slides
  - The slides are under the CodeHandouts/TreeADT/slides folder in the handout repository
  - https://github.com/cs1501-2231/slides-handouts

BST: delete operation

- Deleting an item requires first to find the node with that item in the tree
- Let's assume that we have already found that node
- The method below returns a reference to the root of the tree after removing its root

```
private BinaryNode<T> removeFromRoot(BinaryNode<T> root){
```



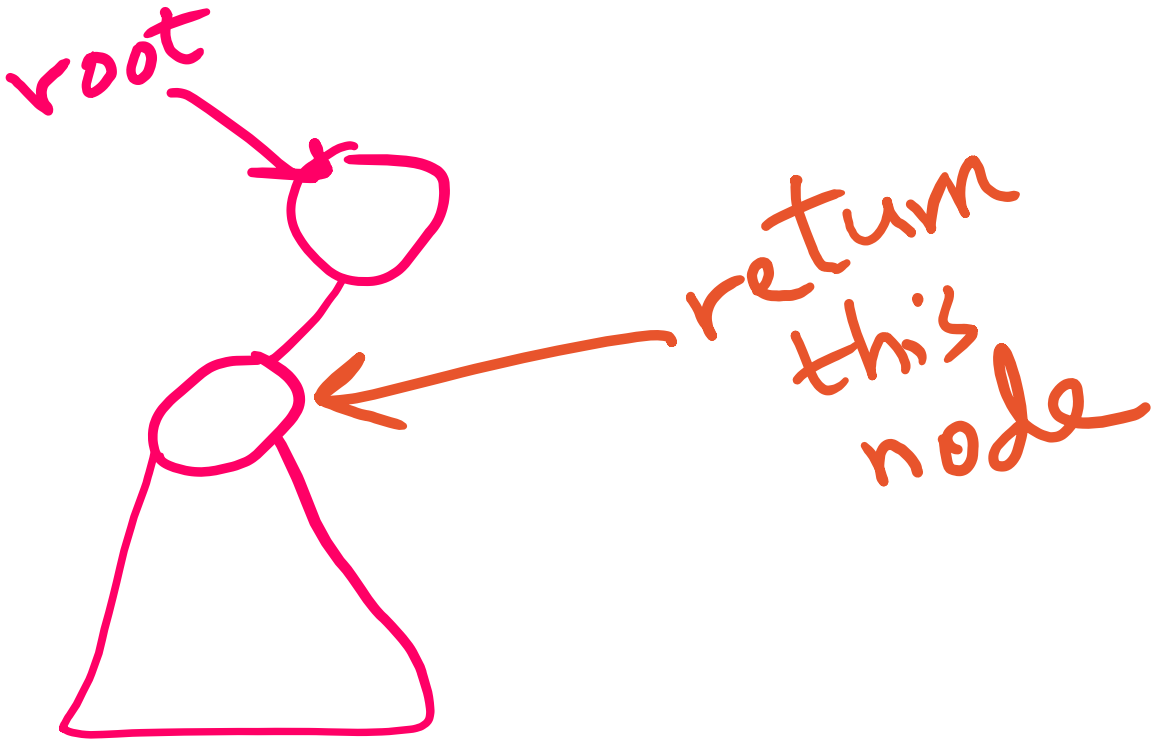
Delete Case 1: tree has only one node

```
private BinaryNode<T> removeFromRoot(BinaryNode<T> root){
```

Return null

Delete Case 1: root has one child (left or right)

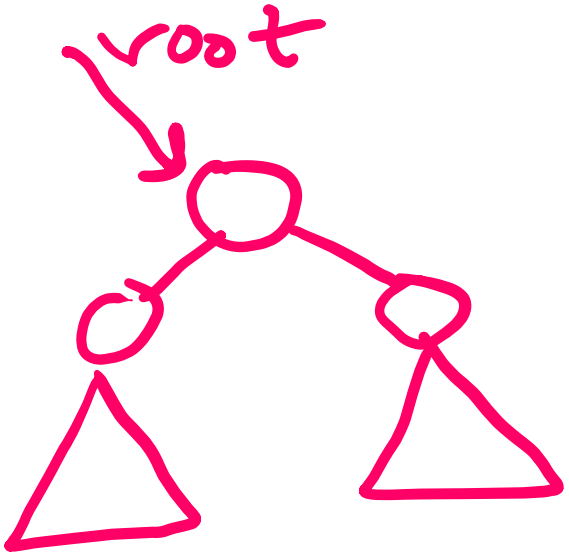
```
private BinaryNode<T> removeFromRoot(BinaryNode<T> root){
```



Return the root of the subtree rooted at the child

Delete Case 1: root has two children

```
private BinaryNode<T> removeFromRoot(BinaryNode<T> root){
```



- replace root's data by the data of the largest item of its left subtree (why?)
- remove the largest item from the left subtree
- return root

How to find largest item in a BST?

```
private BinaryNode<T> findLargest(BinaryNode<T> root){
  if(root.hasRightChild()){
    return findLargest(root.getRightChild());
  } else {
    return root;
  }
}
```

How to remove largest item in a BST?

- The method below returns the root of the tree after deleting the largest item
- If the largest item is the root of the tree, return its left child

```
private BinaryNode<T> removeLargest(BinaryNode<T> root){
  if(root.hasRightChild()){
     root.setRightChild(removeLargest(root.getRightChild()));
  } else {
    root = root.getLeftChild();
  }
  return root;
}
```

Now we need to find the node to delete

- The method below returns the root of the BST after removing the node that contains entry if found
- We also need to return the removed data item
  - How to return two things?
  - Pass a wrapper object

```
private BinaryNode<T> removeEntry(BinaryNode<T> root,
                                  T entry, ReturnObject item){
```

Wrapper Class

```
private class ReturnObject {
  T item;
  private ReturnObject(T entry){
    item = entry;
  }
  private void set(T entry){
    item = entry;
  }
  private T get(){
    return item;
  }
```

Runtime of BST operations

- Search miss, search hit, add
  - O(depth of node)
  - Worst-case: O(n)
  - Average-case: O(log n)
- Delete
  - Finding the node: O(log n) on average
  - Finding and removing largest node in subtree: O(log n) on average
  - Total is O(log n) on average
    - and O(n) in worst-case

Runtime of BST operations

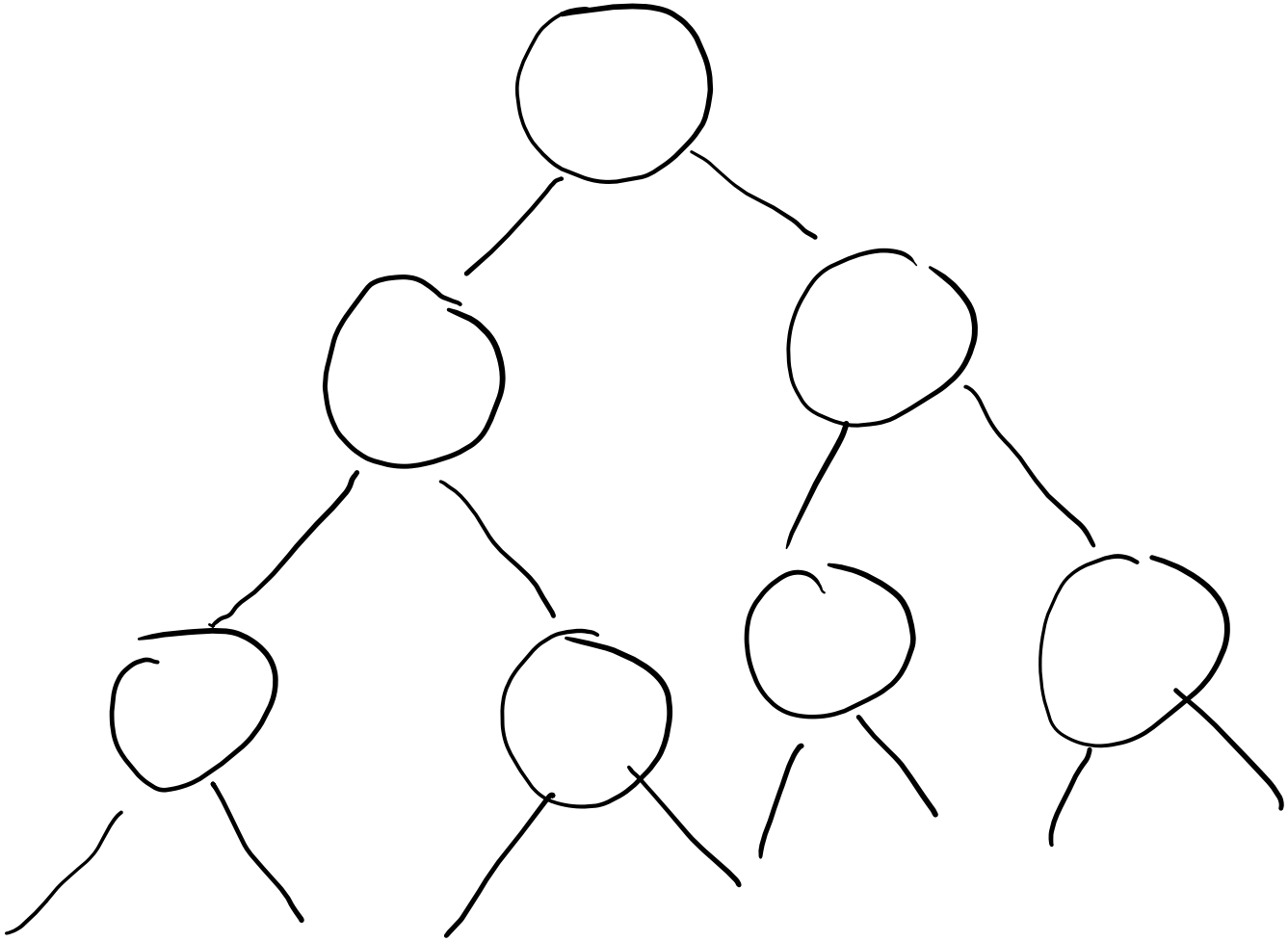
- Can we make the worst-case runtime O(log n)?
- Yes, if we keep the tree *balanced*
  - That is, the difference in height between left and right subtrees is controlled

Red-Black BST

- Definition
  - two colors for edges: red and black
  - a node takes the color of the edge to its parent
  - only left-child edges can be red
  - at most one red-edge connected to each node
  - Each leaf node has two black null-edges out of it (to the two null references)
  - all paths from root to null-edges have the same number of black edges
  - root node is black
  - ***Why?***
    - ***<u>maximum</u> height = 2*log n***
- Basic operations
  - rotate left
  - rotate right
  - flip color
  - ***preserve the properties of the Red-Black BST!***

Red-Black BST example

- All black nodes → has to be a full tree
- Height = O(log(n))

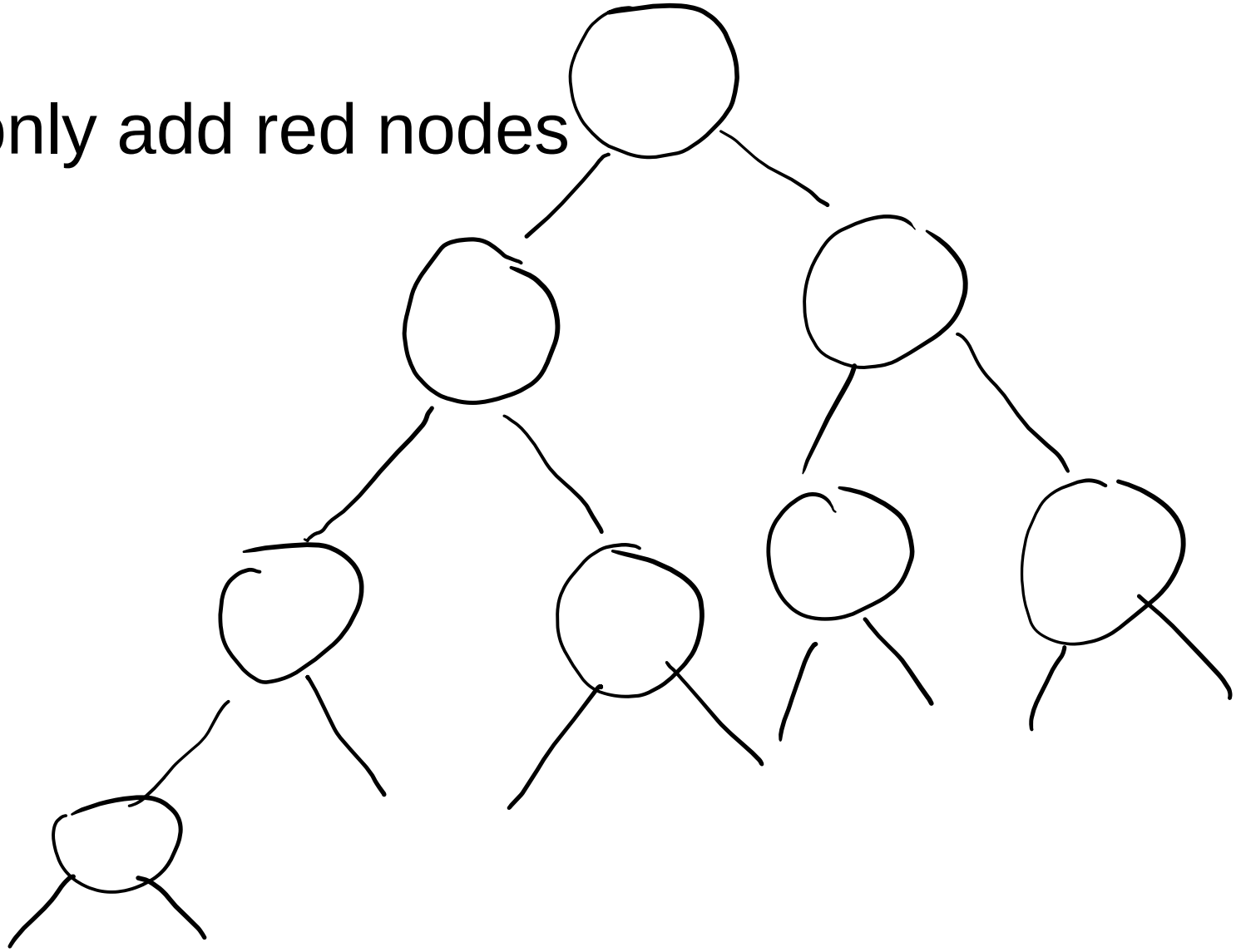


Red-Black BST example

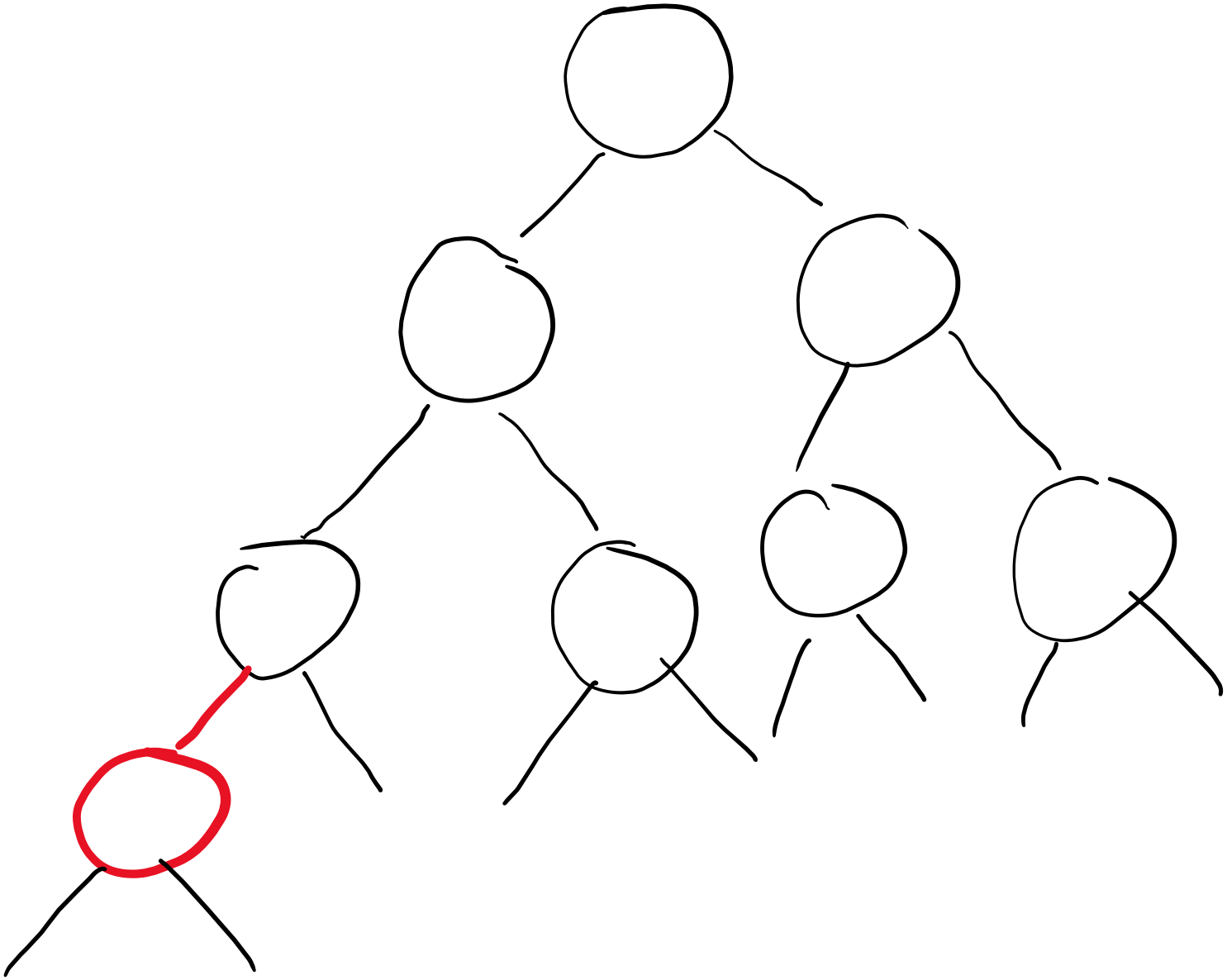
- Let’s imagine an adversary who wants to increase the height of the tree by adding the fewest number of node

Red-Black BST example

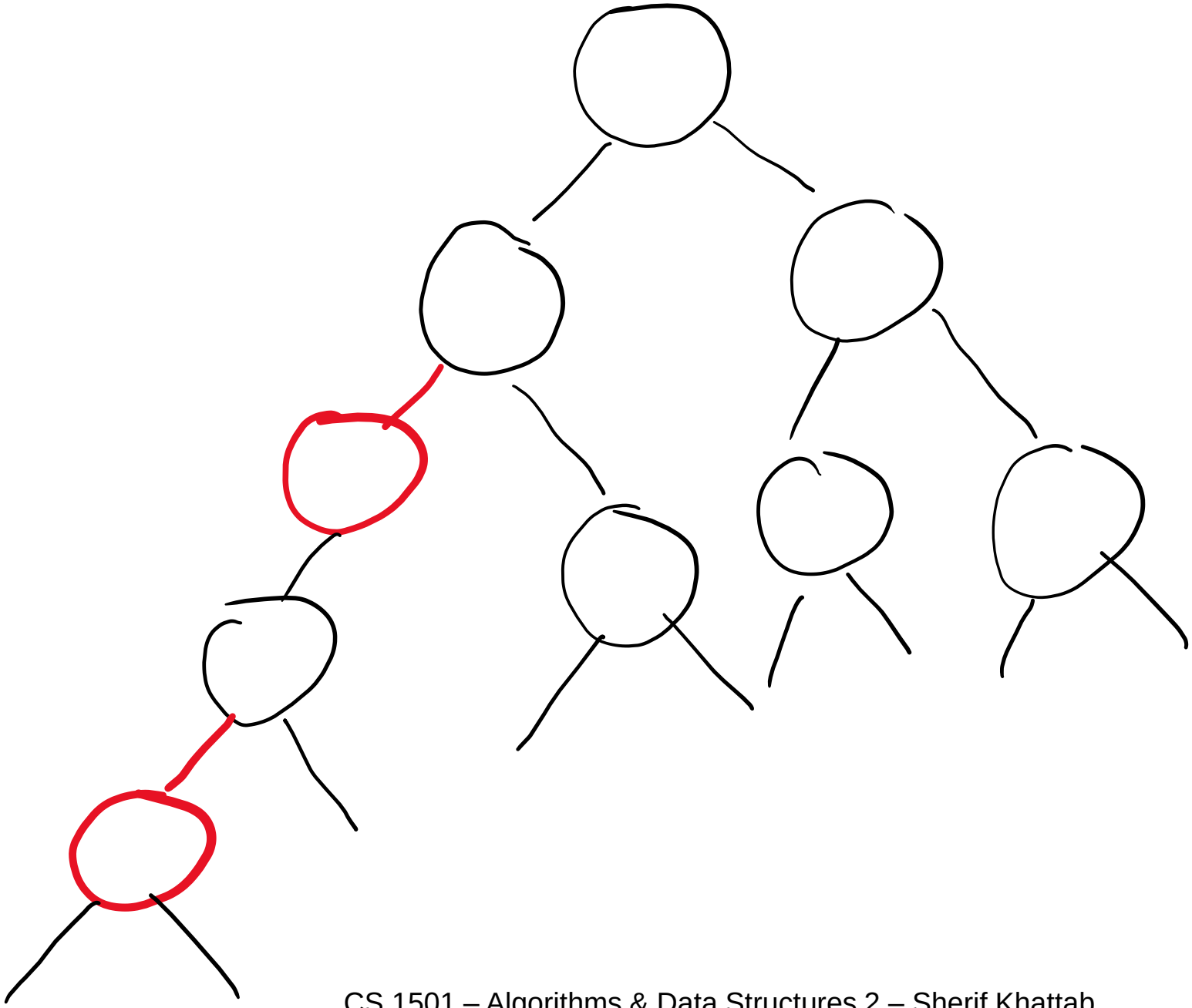
- Can the adversary add a black node?
- No! why?
- They can only add red nodes



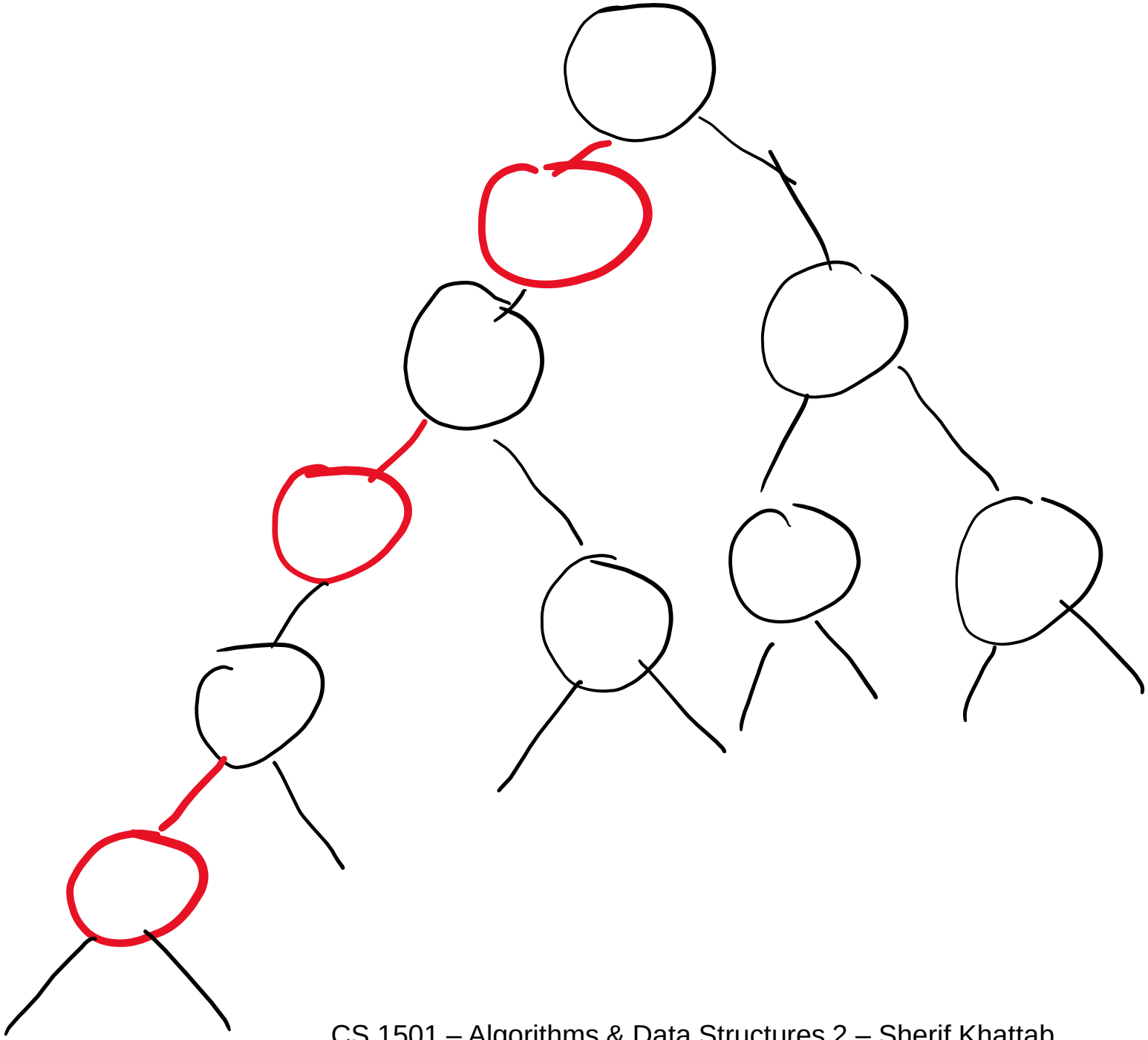
Red-Black BST example



Red-Black BST example

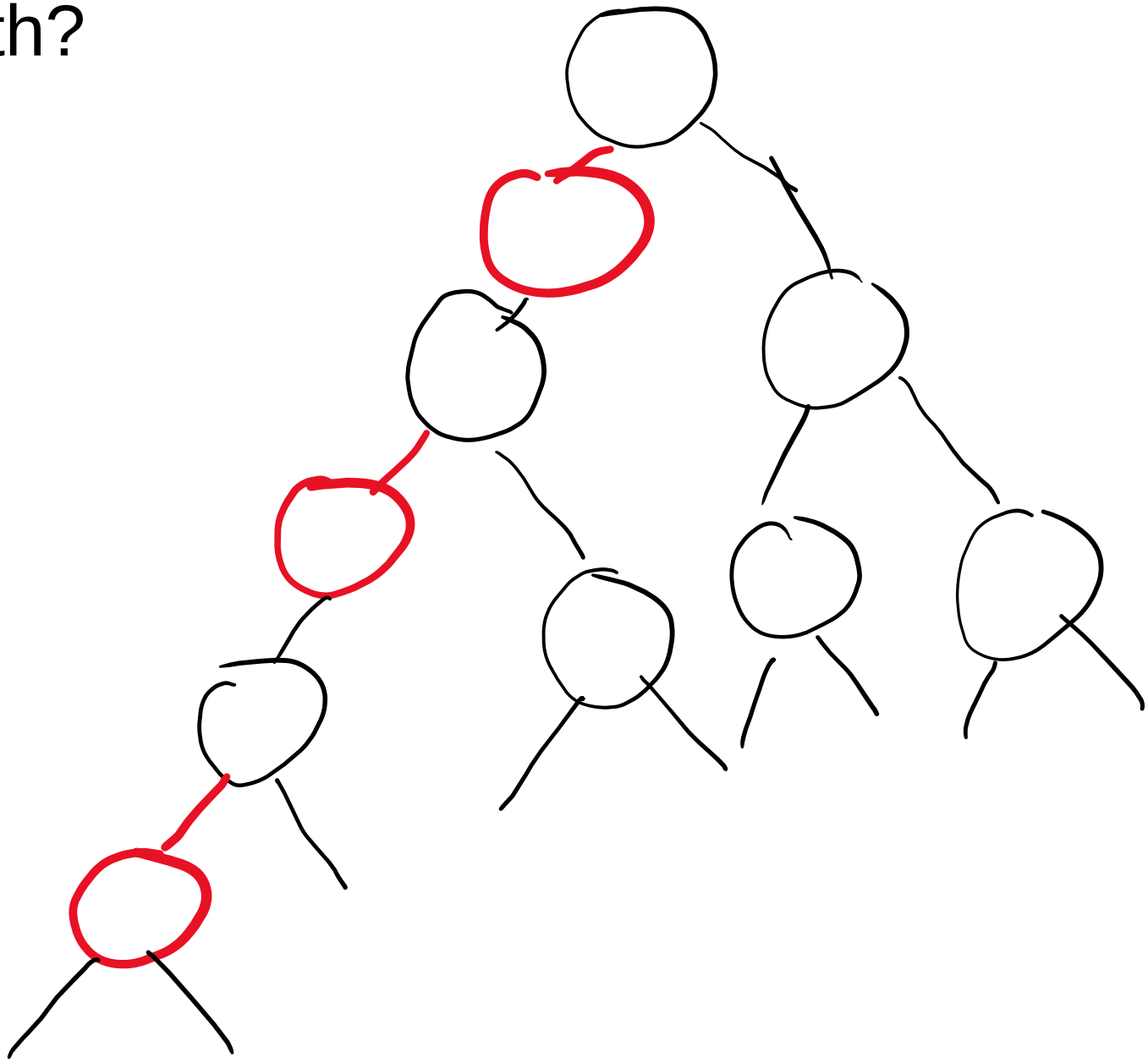


Red-Black BST example



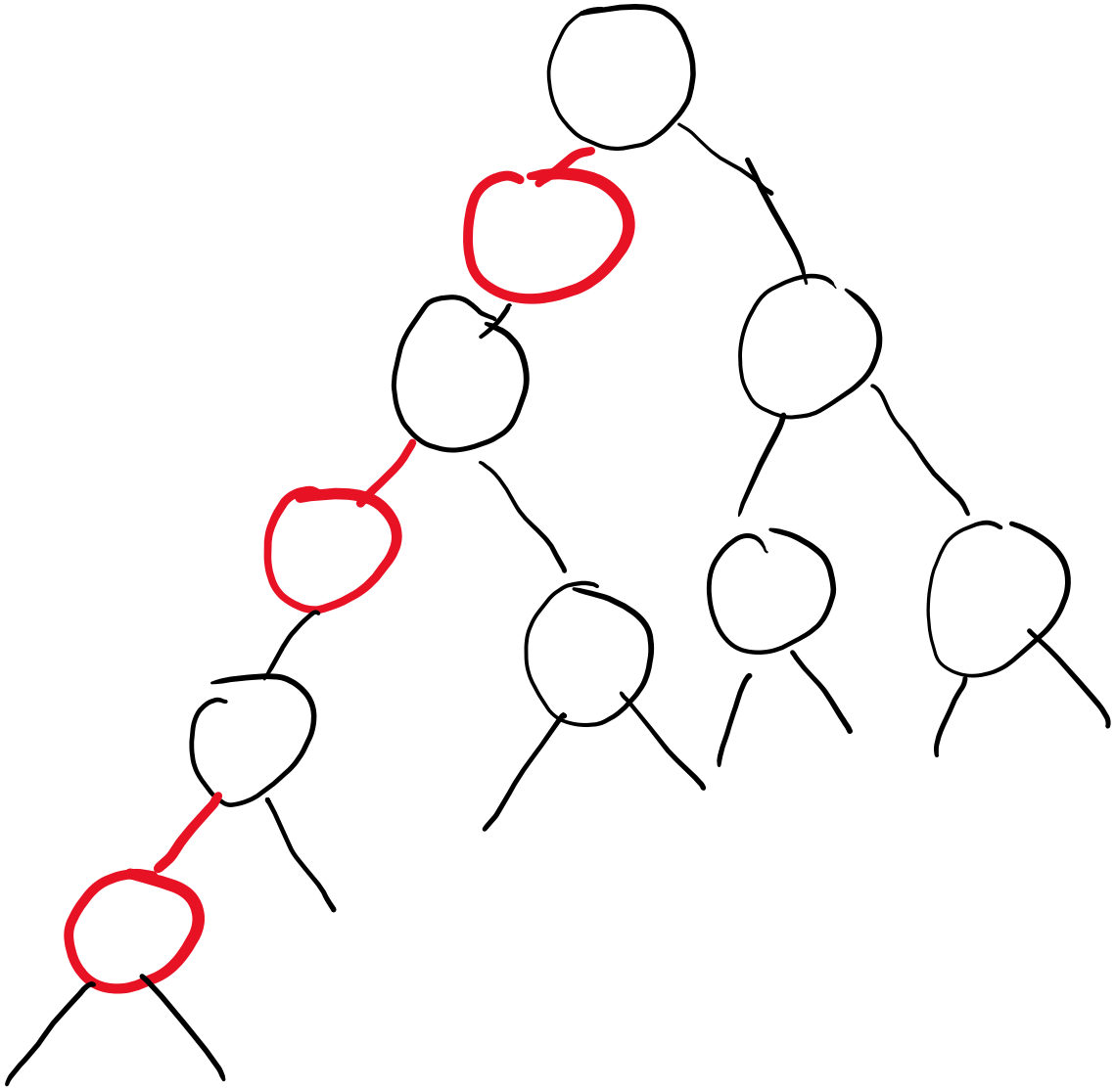
Red-Black BST example

- Can the adversary add more red nodes to the left-most path?

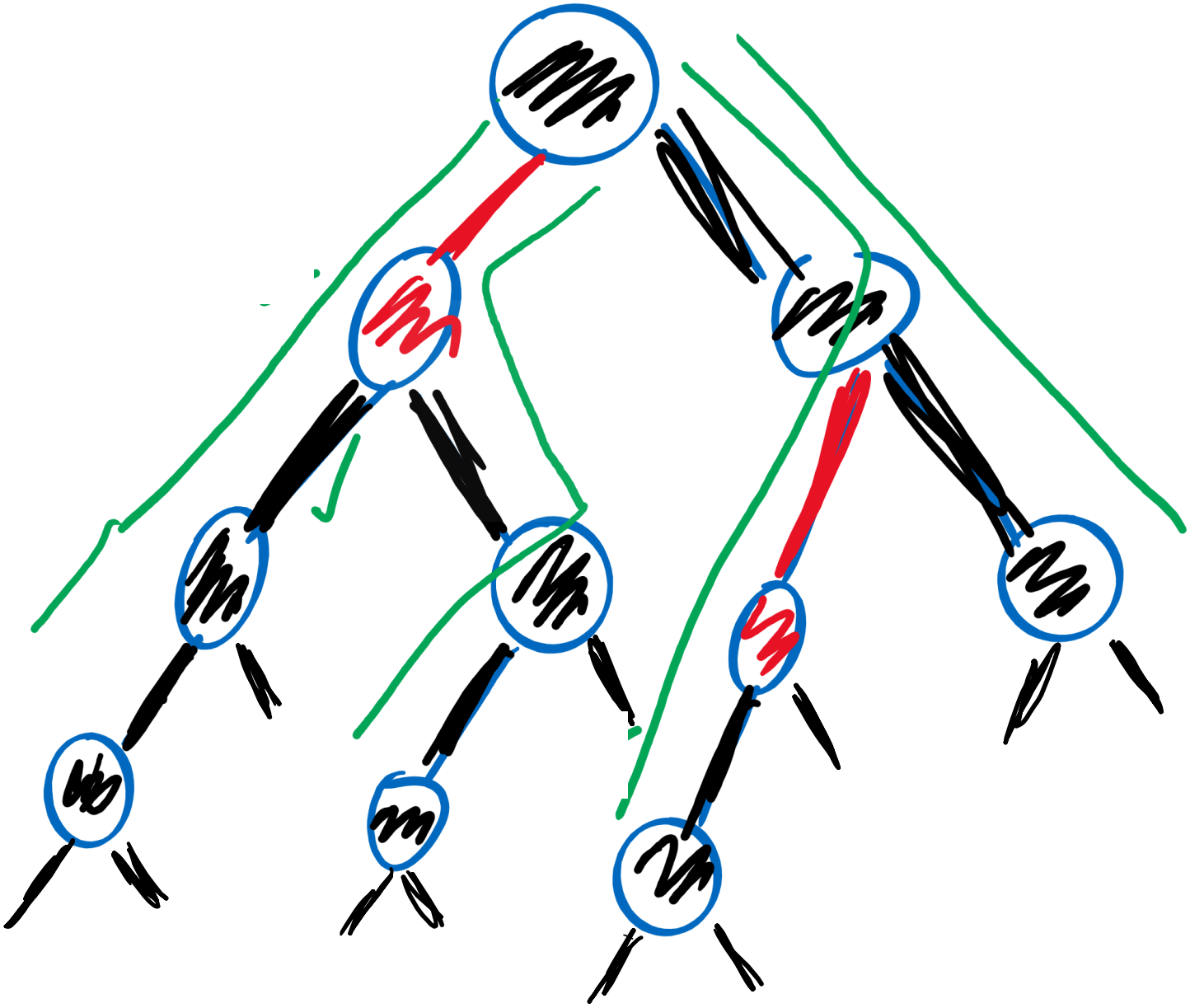


Red-Black BST Example

- The maximum “damage” that the adversary can do is to double the height of the full tree
  - 2* log (n)
  - still O(log n)



Red-Black BST non-example

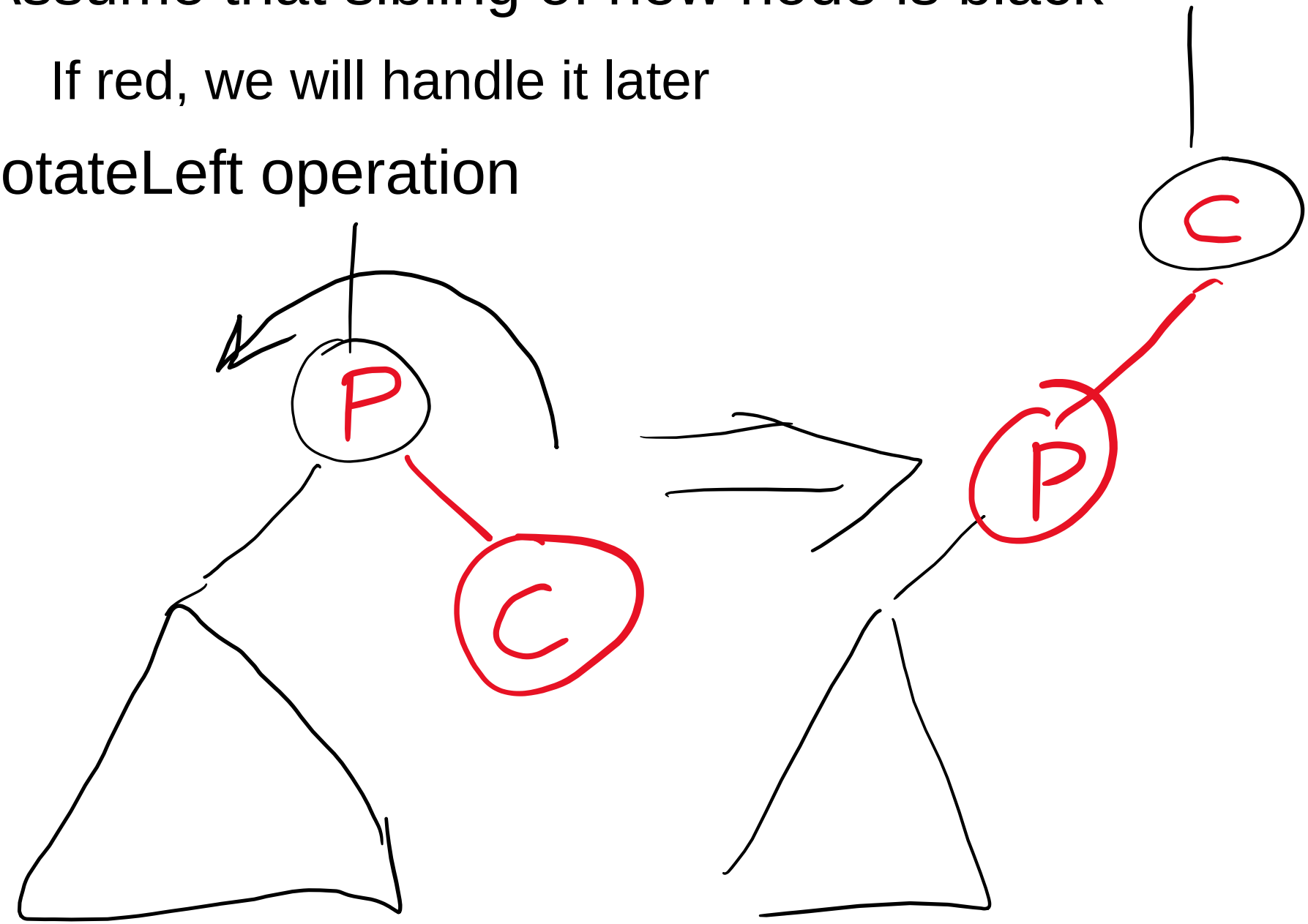


Adding to a RB-BST

- Ok, so we add a red leaf node!
- What can go wrong then?
  - The new node is a right child
  - Fix it by left rotation

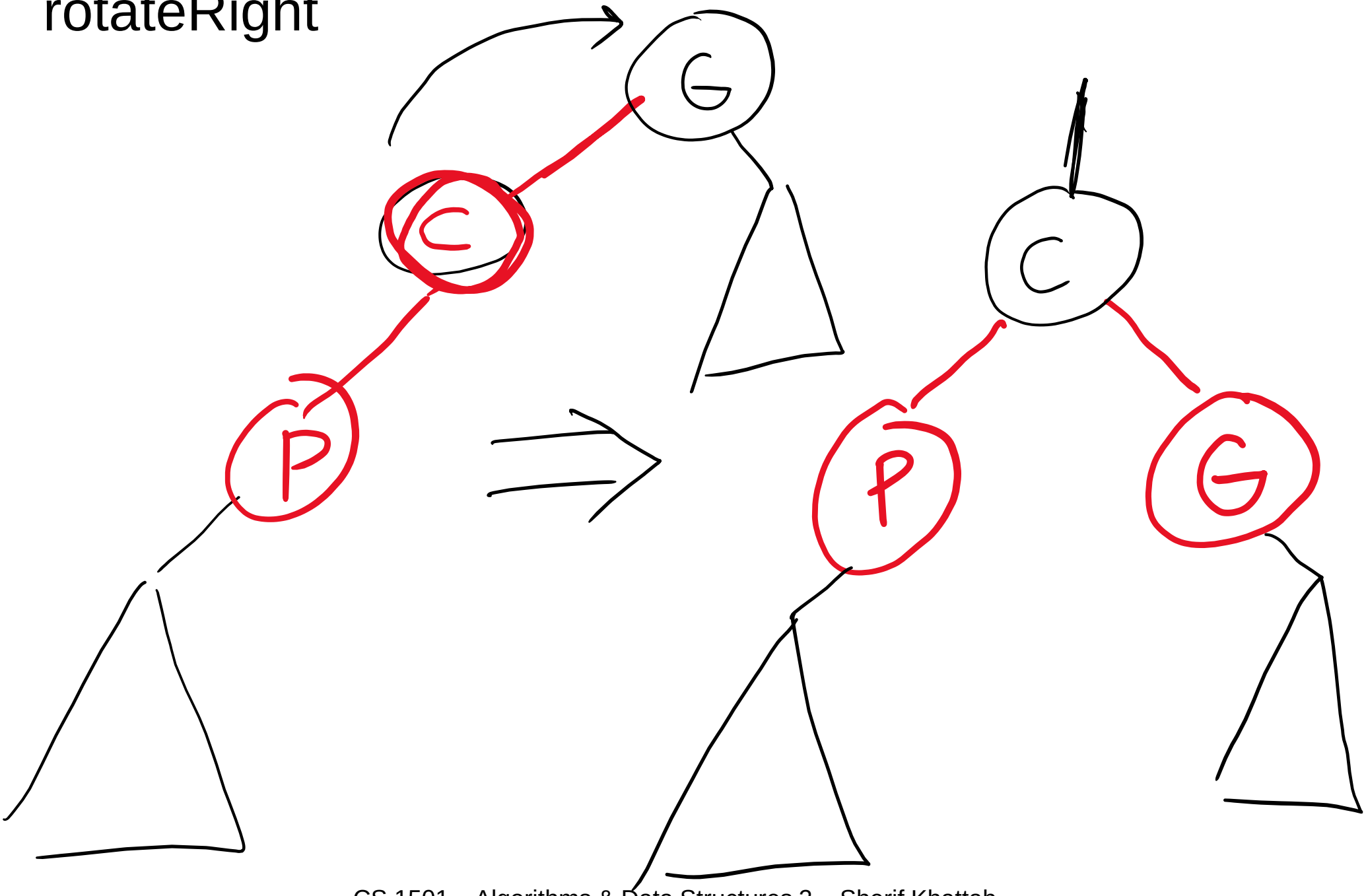
What if the new red node is a right child?

- Assume that sibling of new node is black
  - If red, we will handle it later
- rotateLeft operation



What if the new node becomes red?

- rotateRight



What if both children of a node are red?

- flipColors()